NPS52-89-047

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AN EMPIRICAL COMPARISON OF SOFTWARE FAULT TOLERANCE AND FAULT ELIMINATION

Timothy Shimeall
Nancy Leveson

July 1989

Prepared for:

Naval Postgraduate School
Monterey, California 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared for the Naval Postgraduate School and funded by the National Science Foundation, NASA, CA State Micro Program, and the Naval Postgraduate School Research Council.

Timothy J. Shimeall
TIMOTHY J. SHIMEALL
Assistant Professor
of Computer Science

Reviewed by:

Robert B. McGhee
ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

K. T. Marshall
KNEALE T. MARSHALL
Dean of Information
and Policy Science

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NPS52-89-047	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: University of Calif., Irvine and Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 52	7a. NAME OF MONITORING ORGANIZATION: NAVAL POSTGRADUATE SCHOOL
6c. ADDRESS (City, State, and ZIP Code): Irvine, CA 92717 and Monterey, CA 93943, respectively	7b. ADDRESS (City, State, and ZIP Code): Monterey CA 93943
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: NASA/NSF/CA State Micro/NPS; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: NASA Grant NAG-1-668, NSF Grant DCR-8521398 O&MN, direct funding
8c. ADDRESS (City, State, and ZIP Code): NASA Langley, VA; NSF Washington DC; Micro Sacramento, CA; NPS Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

11. TITLE (Include Security Classification)
AN EMPIRICAL COMPARISON OF SOFTWARE FAULT TOLERANCE AND FAULT ELIMINATION(U)

12. PERSONAL AUTHOR(S)
SHIMEALL, Timothy J. & LEVESON, Nancy G.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Progress	FROM 86/03 TO 89/07	1989, July	48

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Assertions, Back-To-Back Testing, Code Reading, Evaluation of Software Methodologies, Fault Elimination, Fault Tolerance, Testing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Reliability is an important concern in the development of software for modern systems. Some researchers have hypothesized that particular fault-handling approaches or techniques are so effective that other approaches or techniques are superfluous. The authors have performed a study that compares two major approaches to the improvement of software, software fault elimination and software fault tolerance, by examination of the fault detection obtained by five techniques: run-time assertions, multi-version voting, functional testing augmented by structural testing, code reading by stepwise abstraction, and static data-flow analysis. This study has focused on characterizing the sets of faults detected by the techniques and on characterizing the relationships between these sets of faults. The results of the study show that none of the techniques studied is necessarily redundant to any combination of the others. Further results reveal strengths and weaknesses in the fault detection by the techniques studied and suggest directions for future research.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Shimeall, Timothy J.	22b. TELEPHONE (Include Area Code): (408) 646-2509	22c. OFFICE SYMBOL: 52Sm

An Empirical Comparison of Software Fault Tolerance and Fault Elimination[1]

Timothy J. Shimeall

Code 52Sm
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Nancy G. Leveson

Department of Information and Computer Science
University of California, Irvine
Irvine CA 92717

Abstract

Reliability is an important concern in the development of software for modern systems. Some researchers have hypothesized that particular fault-handling approaches or techniques are so effective that other approaches or techniques are superfluous. The authors have performed a study that compares two major approaches to the improvement of software, software fault elimination and software fault tolerance, by examination of the fault detection obtained by five techniques: run-time assertions, multi-version voting, functional testing augmented by structural testing, code reading by stepwise abstraction, and static data-flow analysis. This study has focused on characterizing the sets of faults detected by the techniques and on characterizing the relationships between these sets of faults. The results of the study show that none of the techniques studied is necessarily redundant to any combination of the

[1]This work has been partially supported by NASA Grant NAG-1-668, NSF Grant DCR-8521398, a MICRO grant co-funded by TRW and the State of California, and the Naval Postgraduate School Research Council

others. Further results reveal strengths and weaknesses in the fault detection by the techniques studied and suggest directions for future research.
Key words: Assertions, Back-To-Back Testing, Code Reading, Evaluation of software methodologies, Fault elimination, Fault tolerance, N-version programming, Software experiments, Static analysis, Testing

1 Introduction

Reliability is an important concern in the development of software for modern systems. Software reliability improvement techniques may be classified according to the approach they use to deal with faults (source-code defects): 1) fault-avoidance techniques attempt to prevent the introduction of faults into software during development; 2) fault-elimination techniques attempt to locate and remove faults from software prior to its use; 3) fault-tolerance techniques attempt to prevent faults from causing a program to fail.

Some researchers have hypothesized that particular fault-handling approaches or techniques are so effective that other approaches or techniques are superfluous. It seems important to investigate these hypotheses and, in general, to study the relationship between techniques that use different fault-handling approaches. Such studies provide direction for improving the techniques and for selecting a set of techniques for use on a particular project.

Research is continuing into ways to evaluate and improve various software fault-handling techniques. However, the empirical studies examining these techniques have largely compared them solely on the number of faults detected, rather than

examining the relationship between the sets of faults detected by the techniques. An experiment that examines the intersection of the sets of faults detected by various techniques may reveal new limitations of these techniques, suggest research directions to extend their utility, and provide comparative information.

The authors have performed such a study, contrasting the fault detection capability of a variety of software fault-tolerance and fault-elimination techniques including n-version programming, run-time assertions, functional testing augmented by structural testing, code reading by stepwise abstraction and static data-flow analysis. These techniques were selected for both practical and theoretical reasons. From a theoretical point of view, each of these techniques follow different approaches to the detection of faults, so a broad examination of fault detection is achieved. From a practical point of view, each of these techniques is relatively easy to use, and each technique is within the capability of undergraduate student participants. Each of these techniques have also been examined in previous empirical studies, so comparison of the results of this study with the results of other researchers is possible.

One of the techniques that have been studied empirically is n-version programming. These studies have often revealed limitations in this technique. Despite discouraging empirical results, n-version programming is gaining acceptance among software developers: n-version programming has been used in software to control aircraft[30] and railroads[18] and has been proposed for nuclear power plants[17]. One drawback to the multiversion technique is that the total development costs are increased due to the costs of developing multiple versions. Since in most situations there is a fixed amount of resources that can be invested in building the software, there must be saving somewhere else in order to allow multiple-version development.

In order to make the technique affordable, it has been suggested that n-version programming will be so effective that it can be used as a partial substitute for current software verification and validation procedures[3]. Instead of extensive V&V of a single program, Avizienis and Kelly[4] suggest that independent versions can be executed in an operational environment and V&V completed while using the software. Furthermore, by employing hobbyist programmers to write multiple software versions for critical systems at home, they suggest it will be possible to relax the need for rigorous quality control and centralized programming environments. Although it is difficult to believe that anyone would take these latter suggestions seriously, the need to reduce costs when developing multiple versions has led to at least one attempt to reduce testing of safety-critical software for commercial aircraft based on the argument that the use of n-version programming provides such high reliability that unit testing can be reduced. But the hypothesis that testing can be reduced in multiversion software needs to be carefully examined, both empirically and theoretically, before modifying current software development practices. No such evaluation has yet been performed.

The goal of our study is to compare some fault elimination and fault-tolerance techniques in terms of faults detected in order to provide data on the issues and questions raised by previous experiments. After a survey of the relevant research, this paper describes the experimental procedure used in our study, characterizes the results and summarizes the impact of those results in providing direction for future research.

2 Related Work

2.1 Fault-Elimination Experiments

There are a large number of studies examining software testing. Much of the recent work has focused on assessing the effectiveness of various testing techniques. Myers[27] did a comparative study of functional testing against code reading for fault detection in small FORTRAN programs. The code-reading methodology used was an informal desk check conducted by 3 participants. Myers found a wide variation between individuals, but no significant difference between the performance of the two techniques.

A study by Hetzel[19] compared code reading, structural testing and functional testing in terms of the faults detected by each technique. In that study, 39 experienced subjects tested three PL/I programs ranging in length from 64 to 170 statements. The programs were a sequential series of nested data transformations. The code-reading technique used in the study, developed by Hetzel, exploited this structure to summarize the effect of the programs by summarizing the effect of each transformation. The structural-testing criterion used was statement coverage. The study found that functional testing discovered the most faults and code reading the least, with structural testing falling in between. Code reading detected faults for which test cases are hard to derive, and it detected initialization faults.

Basili and Selby[6] compared code reading by stepwise abstraction with functional and structural testing in four small programs (145 to 365 lines long) written in an Algol-like language. Three of the programs contained naturally-occurring faults, while the fourth contained a mixture of naturally-occurring and seeded faults. The

structural-testing criterion used was statement coverage. Functional testing used equivalence partitioning and boundary-value analysis.

In the Basili and Selby study, code reading by stepwise abstraction detected more faults than either of the other techniques studied. Structural testing detected fewer faults than functional testing. The study also compared the types of faults detected by each method using two classification schemes: omission vs. commission and the type of operation in which the fault was present (initialization, control, data, computation, interface or output). Code reading and functional testing detected insignificantly different numbers of each class of faults except interface faults, where code reading detected significantly more, and control faults, where functional testing detected significantly more. In each case, the structural testing detected either significantly less, or insignificantly different numbers of faults of each type. The study did not provide information on the size of the intersections of the sets of faults detected by each method nor on the sets of faults detected by combinations of methods.

Girgis and Woodward[16] compared the fault-detection abilities of four types of testing: weak mutation testing, data-flow testing, control-flow testing and static data-flow analysis. The comparison used a set of small FORTRAN programs (text-book examples) that were seeded with faults one at a time by an automated tool, then tested until either the seeded fault was detected or the testing criteria were satisfied. The results indicate a large variation in the effectiveness of the testing criteria. Analysis of the experimental data shows an insignificant difference between the four groups, due to the large variation between the criteria in each group. The differences between the individual criteria were significant. The most effective crite-

rion was All-LCSAJs (Linear Code Sequence and Jump). However, this study failed to indicate if this effectiveness result was due to the choice of faults seeded in the programs or to characteristics of the detection techniques (e.g., the seeding strategy may have favored All-LCSAJs). Moreover, the results may have been influenced by the particular testing strategies of each type used.

Ramamoorthy and Ho[29] studied two forms of static data-flow analysis on large FORTRAN programs. Their results confirmed the limitations of static data-flow analysis, but faults were detected during their experiment. In the 2,000 line program analyzed in that study, simple static data-flow analysis (analysis for unreachable code, interface inconsistencies and locally-uninitialized variables) detected four faults. In a separate 23,000 line program, a more comprehensive static data-flow analysis (performing more thorough uninitialized-variable checking, loop-increment checking and analysis for branch anomalies) detected 20 faults.

There have been proposals to use multi-version voting in the testing process [7, 8, 28, 31]. In this method, known as back-to-back testing, the vote itself is used as the test oracle, and therefore a larger number of tests can be executed. The underlying assumptions here are that (1) given that a fault leads to an erroneous output it will be detected by the voting process, and (2) the faults that would have been detected by other testing techniques, such as structural testing or static analysis techniques, will be elicited and detected by voting on random or functional test cases alone.

A study by Bishop et. al.[7] examined back-to-back testing by varying the specification language, development practices and implementation language used for the versions. Three professionally-developed versions were used and seven faults were

detected by back-to-back testing, after an initial acceptance test. Of the seven faults, two were common between two of the three versions used. No independent method of verification of the results was used so three-way failures could not be detected. However, these results are difficult to interpret due to two factors. The experiment used a small number of versions, which causes the results to be strongly affected by characteristics of the individual versions, with no capability of identifying what characteristics are due to the different techniques that produced the versions. The experiment also lacked isolation between independent variables. Two of the versions were coded in the same implementation language (FORTRAN 77). One of these two and the third version used a common specification (jointly prepared), but all three used different design methods. This design cannot distinguish between the effects of varying design methods and the effects of different language-specification mixtures.

Given the contradictory results of previous studies, it appears that no simple rules exist for choosing among these testing techniques. Furthermore, while relative comparisons of number of errors detected provide some basis for choosing between mutually exclusive alternatives, this is not necessarily the situation with respect to testing. Although limited resources and time usually forces limitations in the total amount of testing performed, one would probably want to apply more than one method for detecting software faults. It would be helpful to have information on the degree to which two techniques are complementary, i.e., detect different errors, or redundant, i.e., detect the same errors, along with more detailed information about the particular errors (and hopefully classes of errors) detected and not detected. Some of this information can be derived by theoretical analysis while some will

require empirical study since human behavior and capabilities are involved for which few adequate models exist.

2.2 Fault-Tolerance Experiments

There have been very few experiments that have explored the use of assertions for fault tolerance. A study by Anderson[2] applied recovery blocks, which use assertions to test the system state, to a real-time control system. The code (a professionally-implemented version of a submarine-control program) was 8000 lines long and organized into 14 concurrent activities. The results showed that while assertions were quite difficult to formulate, some reliability improvement was gained through the use of recovery blocks.

Using software from a voting experiment[22], Leveson, Cha, Knight and Shimeall [24] had a set of 24 students instrument eight versions of a Pascal program (varying in length from 400 to 800 lines). These versions were all known to produce correct results in excess of 99.5% of the time. The investigators had the participants plan their augmentation of the source code working strictly from the specification and then proceed to augment the code with the planned assertions, plus any other assertions the participants desired. The investigators tested the 24 instrumented programs using random data as well as data on which the original versions were known to fail. The assertions detected only 14 of the 60 known faults in the 24 instrumented versions. However, assertions detected 6 additional faults (not previously known). Examining the cases where known faults were not detected revealed three causes for ineffective checks: bad check placement, bad check condition, and use of faulty code from the original version in the check. These results show that

assertions are effective in detecting faults even in software of relatively high quality. In this study, the assertions found as many faults as voting (but largely different faults). Lastly, the results showed that specification-based checks alone were not as effective as using them together with code-based checks.

There have been several experiments involving the use of n-version programming. The first, by Chen[12], provided little information because of difficulties in executing the experiment. However, it was noted that 10% of the test cases caused failures for the 3-version systems (35 failures in 384 test cases). Chen reported that there were several types of design faults that were not well tolerated in this experiment, in particular missing-case logic faults.

Avizienis and Kelly[4] examined the use of multiple specification languages in developing multi-version software. The reported data indicates that in over 20% of 100 test cases executed, the three version-systems were unable to agree or voted on a wrong answer. In addition, 11 of the 18 programs aborted on invalid input. Despite these results, they conclude that "By combining software versions that have not been subjected to V&V testing to produce highly reliable multiversion software, we may be able to decrease cost while increasing reliability." The data in this experiment does not support the hypothesis implicit in this statement that high reliability will be achieved by using this technique on software that has not been rigorously tested. They continue to say that "Most errors in the software versions will be detected by the decision algorithm during on-line production use of the system." There were 816 combinations of the programs in this experiment each run on 100 test cases for a total of 81,600 calculated results. In 5.6% of the cases where an error occurred in at least one version, the error was not detected by the

voting procedure, i.e., the programs agreed on a wrong answer.

Another experiment, by Knight and Leveson[23], found that with 27 programs run on 1,000,000 test cases, an error was not detected by voting three versions in 35% of the cases where an error actually occurred. The individual programs in this experiment had a much higher average reliability than in the Kelly experiment (i.e., 0.9993 versus 0.72) indicating that they were more thoroughly tested before being subjected to the voting procedure. The results provide some justification for hypothesizing that faults leading to correlated failures are more likely to escape detection during testing than faults that do not lead to correlated failures.

Knight and Leveson[22] investigated the problems of common failures between independently-produced versions and have also looked at reliability improvements [23]. Although the failure probability was decreased (about 10 times) using three-version voting compared to single versions in the latter study, this comparison may not be a realistic one. It is reasonable to expect that applying some reliability-enhancing technique would produce an improvement over not applying any special techniques. A more realistic comparison is to examine the reliability of multiple versions voted together versus the reliability of a single version with additional reliability-enhancement techniques applied.

Although it was not the original goal, there is a study that provides one data point in this comparison. Brunelle and Eckhardt[11] took a portion of the SIFT operating system, which was written from a formally-verified design at SRI[26], and ran it in a three-way voting scheme with two new (non-formally verified) versions. The results showed that although no faults were found in the original SRI version, there were instances where the two unverified versions outvoted the correct, verified

version to produce a wrong answer[2]. Care must be taken in using this data because the qualifications of the implementors of the three versions may be different.

A study by Avizienis, Liu and Shütz[5] tried to examine the use of multiple implementation languages in n-version programming. Six versions of a flight-control program were implemented separately (one in each of six languages) and executed in triplets to look for faults. It is difficult to use the results of this experiment to understand n-version programming, since a detailed design specification (including the algorithms to be used and the form of the internal state) was included as part of the problem specification. Since there was minimal design diversity, there was unlikely to be detection of design errors, only coding errors or typographical errors. For example, two of the versions were found to be erroneous because a hand-written value of 65,536 was misread as 65.536. Furthermore, the versions were evaluated to reach agreement among themselves without any use of a separate standard of correctness. After a carefully-controlled development involving reviews of the versions for discrepancies, the investigators found few functional differences between the versions (which is not surprising given that the programmers were all given the same design specification).

Examining the results obtained by the previous experiments reveals several characteristics of n-version programming. First, the prevalence of coincident failures (observed in every experiment conducted thus far) reduces the effectiveness of multi-version voting in dealing with faults. Second, there appears to be substantial difficulty in getting versions to agree on a consensus result. Research has found that even mathematically correct algorithms sometimes produce differing results due to

[2]These results are not reported in the published paper on the experiment, but were obtained through personal communication with one of the authors.

numeric instability[9]. Agreement concerns were cited in the decision by Avizienis, Liu and Shütz[5] to specify the algorithm to be used in their versions. However, this solution essentially sacrifices the idea of version or design diversity, by limiting it to the syntactic level, which may have only minor effect on software reliability.

Finally, analysis of the experiments shows that multi-version voting is a fault-specific technique. Decisions on the portion of the system to be written in a multi-version manner and on what algorithms are to be used in the different versions limit the class of faults that these techniques are able to handle. The known problems in handling missing-case logic faults indicate this specificity. What distinguishes this technique from other fault-specific techniques is the lack of an explicit characterization of what faults are handled.

Much of the existing research on n-version programming has employed uncontrolled experimental designs. Claims have been made of improvements in reliability due to these techniques in comparison with unverified software. This comparison is unrealistic. The alternative to using multi-version voting is not to leave the software unverified, but to apply other verification and validation techniques.

3 Experimental Design

A set of programs written from a single specification for a combat simulation problem are used in the study described in this paper. The specification is derived from an industrial specification obtained from TRW[13]. The simulation is structured as three sets of transformations from the input data to the output data. The first set of transformations converts the input data to an abstract intermediate state. The

intermediate state is updated by a second set of transformations in each cycle of simulated time. After a number of cycles (specified in the input data), the output data are produced by the third set of transformations from the final intermediate state. Prototype implementations were developed by three individuals in order to evaluate and improve the quality and comprehensibility of the requirements specification before the development of the versions began.

The experiment participants used throughout were upper-division computer science students. One set of participants, students in a senior-level class on advanced software engineering methods, performed all design and implementation activities on the program versions. A disjoint set of participants attempted to detect faults in the programs. All decisions on whether or not to report a section of code as a fault were made by a student participant or a piece of software written by a student participant. Once the reports were generated (by all techniques), the administrator acted as final arbiter as to which reports identified faults and which were false alarms; this decision was not reported to the students during their participation in the experiment. All participants were trained in the techniques used in the experiment; however, none had applied these specific techniques on any projects prior to this experiment with the exception of previous Pascal programming experience by the implementation participants.

The development activity involved 26 individuals, working in two-person teams. Teams were assigned randomly. Of the 13 teams, 8 eventually produced versions that were judged acceptable for use in the experiment. The development activity involved preparing architectural and detailed designs for the software, coding the software from those designs, and debugging the software sufficiently to pass the

version acceptance test. The version acceptance test was a set of 15 data sets. The data sets were designed to execute each of the major portions of the code at least once. The acceptance test was not, and was not intended to be, a basis for quality assessment of the code, but rather was a test of whether all major portions of the code were present in some operable form. The goal of the development procedure was to have the versions in a state similar to that of normal software development immediately prior to unit testing.

Table 1 describes the finished versions. The column marked 'Modules' shows the number of Pascal procedures and functions in each version. The size of the source is given in two figures, source lines and code lines, with the latter figure omitting blank and comment lines. Finally, the complexity of each module in each version was measured using McCabe's Cyclomatic Complexity, V(G). These figures are profiled in the last three columns of the table. The minimum V(G) is omitted since in each version there was at least one module that was strictly linear in structure (no loops or branches), and thus has a V(G) of 1. The mean code length is 1777 lines, with a standard deviation of 435.

The experimental activity involved applying five different fault detection techniques to the program versions: code reading by stepwise abstraction[25], static data flow analysis, run-time assertions inserted by the development participants, multi-version voting, and functional testing with follow-on structural testing. The code reading was performed by eight individuals. Prior to code reading, all developer-inserted comments were stripped from the version source code. Each version was read by one person, and each person read only one version. The data-flow analysis was performed by implementing and executing an analysis tool based on algorithms

#	Version			Module V(G)		
	Modules	Source Lines	Code Lines	Mean	Std. Dev.	Max
1	72	7503	2414	4.671	6.844	47
2	56	3452	1540	3.055	4.125	13
3	41	1480	1201	3.698	5.549	25
4	57	3663	2003	4.825	5.282	21
5	28	1834	1544	7.694	13.011	58
6	72	3065	2206	2.868	3.423	16
7	75	2734	1978	3.427	5.084	28
8	57	1896	1331	2.193	2.503	10

Table 1: Version Source Profile

by Fosdick and Osterweil[14].

The development participants were trained in writing run-time assertions and were required to include assertions in their versions. The run-time assertions were present during the application of all techniques. If an assertion condition fails, a message is generated.

A "gold" version has been written by the experiment administrator as an aid for fault diagnosis, but this actually just provides another version to check against. In fact, faults in the gold version have been detected. The gold version is not included in the experimental data. It is, of course, possible that failures common to all of the versions, including the gold, will not be detected. This is an unavoidable consequence of this type of experiment.

To examine fault tolerance by voting, we chose three versions because more than this number of versions would, in most cases, be totally unrealistic, and two-version voting does not provide fault tolerance. When at least one version in a two-version system provides an erroneous result, there is no chance of masking (or tolerating)

that failure and either no answer or a wrong answer is provided. All versions were executed on a set of 10,000 randomly-generated test cases. The test data generator has been designed to provide realistic test cases according to an expected usage profile in the operational environment.

Functional testing augmented by structural testing was performed on the programs. A series of 97 functional test-data sets were generated from the specification by trained undergraduates. These data sets were planned using the abstract function technique described by Howden[20]. Part of each plan was a description of the program instrumentation needed to view the output of each abstract function. The structural coverage of the functional data-sets was measured using the ASSET structural testing tool[15], and sufficient additional data sets were defined to bring the coverage up to the all-predicate-uses level. The participants used a total of 60 additional data sets to achieve all-predicate-uses coverage in all procedures in all versions. The number of data sets executed by each individual version varied from 5 to 13, as needed to achieve the required coverage.

Because some of the techniques applied to the programs are open-ended in terms of possible application of resources, it was necessary to attempt to hold relatively constant the resources allocated to each technique. This was not necessary for those techniques, namely static data flow analysis and code reading, that have a fixed and relatively low cost. Table 2 contains the amount of human hours and computer hours devoted to each technique. The time devoted to software testing and voting is approximately two calendar months per version for both.

In general, the student participants took their efforts seriously and performed careful work in this experiment. In the code reading, the average rate of analysis

Technique	Computer Hours				Human Hours			
	Mean	SD	Min.	Max.	Mean	SD	Min.	Max.
Code Reading	0	0	0	0	36	15	19	60
Static Analysis	40	30	0.5	104	1	0.1	0.75	1.25
Software Test	84	63	36	219	373	4	366	378
Voting	1415	1055	600	3692	6	1	4	8

Table 2: Hours Devoted to Each Technique Per Version

and annotation was 70 lines per hour. Analysis of the annotations shows that the students were reading in depth and not simply skimming the source listings. In the functional test planning, similar attention to detail was displayed by the students. It is possible for test plans to be of variable quality, so three outside industrial experts evaluated the plans in an attempt to gauge the plan quality before testing the versions. In their opinion the plans, while obviously not the work of experienced professionals, were comparable to those used in many industrial settings.

4 Results

Two general categories of questions have guided our analysis of the data. The first is a comparison between fault elimination and fault-tolerance techniques, i.e., are they substitutes for each other or do they complement each other. The second category of questions involves comparing various testing techniques with respect to fault detection, including consideration of their relative strengths and weaknesses and how these techniques might be improved.

4.1 Fault Tolerance and Fault Elimination

Before presenting the data, it is necessary to define some terms. In a three-version voting system, there are three possible results: a correct answer is produced (there are at least 2 programs in the triplet that produce correct answers), no answer is produced (three different results are produced), or a wrong answer is agreed upon (at least two programs produce identically wrong answers). If a correct answer is produced despite the failure of one of the programs, then the triplet is fault tolerant and the single error is masked. If no agreement is reached, then one could say that the individual program failures were detected, but fault tolerance (run-time masking) has not been achieved. In the third case, i.e., producing an incorrect result, the failure is not detected and the faults have not been tolerated.

To examine whether fault-tolerance techniques are substitutes for fault-elimination techniques, we shall consider the faults tolerated by 3-version voting and potentially tolerated by techniques using assertions. Note that the assertions themselves provide no fault recovery ability, but may be used in conjunction with either forward or backward recovery strategies to tolerate faults once they are detected. No recovery strategies were implemented in the programs used in this experiment. This means that the results relating to assertions should be viewed as counts of faults potentially tolerated (if the hypothetical recovery techniques were effective) rather than faults actually tolerated. This contrasts with fault tolerance by voting, where the same mechanism (a vote) is used to detect and tolerate faults. The results in this section relating to voting are counts of faults actually tolerated by 3-version voting in this experiment.

The first question that was investigated is whether run-time voting tolerated the

faults detected by the fault-elimination techniques used, i.e, functional testing augmented by some structural testing, code reading, and static analysis. This question tests the previously-quoted hypothesis by Avizienis and Kelly[4] that multiversion voting may reduce or replace traditional software V&V.

If voting tolerates the faults detected by testing, then elimination or reduction in testing can possibly be justified, and testing could be completed while the software is being used. However, if the faults detected by fault elimination are *not* tolerated by voting at run-time, then testing cannot be eliminated. Furthermore, any argument for reduction of testing would need to prove that the reduction in testing merely results in the non-detection (and non-elimination) of the faults that voting will reliably tolerate and does not result in the software containing faults causing run-time failures that might have been detected and eliminated by increased testing.

There are two aspects to answering this question. The first is whether the same faults are detected by the fault-elimination techniques and tolerated by voting. If not, then testing cannot be eliminated and reduction of the amount of each type of testing cannot be justified. However, testing can also be reduced by eliminating just one type of testing. Therefore, it is also necessary to investigate whether one particular type of testing is superfluous when using voting. The first aspect is covered in the rest of this section of the paper; the second is discussed in the next section where the faults detected by each technique are compared.

The programs were executed on 10,000 randomly-generated data sets. In general, we found that the faults that were tolerated were not the same as the faults that were detected by fault-elimination techniques. In addition, of the 56 total voting triplets, none tolerated more than a relatively small fraction of the faults detected by the

fault-elimination techniques. The best triplet tolerated only 41 faults (compared to 107 detected by all techniques in the versions (numbers 4, 5 and 6) that participated in that triplet). All triplets were unreliable in tolerating even those faults they could tolerate (i.e., those faults they tolerated at least once).

Table 3 shows the number and intersection of faults found by each class of technique, i.e., voting, assertions and the three fault-elimination techniques. Assertions are included in the results that follow solely to better characterize the relationship between the fault tolerance and fault-elimination approaches. Note the relatively small number of faults that were both tolerated by voting and detected by a fault-elimination technique (27, given by the sum of the last two lines of table 3). It is also interesting to note that assertions and voting detected only a few of the same faults. This supports the results by Leveson, Cha, Knight and Shimeall[24].

A second question involves the relationship between coincident failures and testing. There has been speculation about whether the faults that result in coincident failures (and thus reduce the fault-tolerance capability of voting systems) are likely to be detected through testing procedures. Examination of the specific faults that were detected by testing indicates that testing detected only 24 of the 103 common-failure faults. Furthermore, the common-failure faults found by testing did not include those faults that produced the majority of the common failures during the executions. This result occurred despite the fact that some of the testing techniques target the testing of special cases, which are often involved in common failures. This result may indicate that the faults that reduce the effectiveness of n-version programming are among the most difficult to detect. This is not surprising and satisfies the intuitive explanation that the parts of the problem that lead to mistakes by the

programmers may be equally difficult for the testers to handle. That is, humans are unlikely to make mistakes in a random fashion.

	Version								
	1	2	3	4	5	6	7	8	Total
Tolerated by vote	7	7	11	8	14	7	5	8	67
Detected by assertions only	4	3	1	8	2	2	4	4	28
Detected by fault elim. only	4	17	27	13	15	4	13	26	119
Both assert & fault elim.	7	1	1	3	6	3	0	0	21
Both assert & vote	0	0	2	0	1	0	2	3	8
Both vote & fault elim.	0	2	3	3	2	4	5	2	21
Assert, vote & fault elim.	0	0	1	1	2	1	0	1	6

Table 3: Number of Faults Tolerated or Detected

Finally, it is interesting to ask if there were any faults tolerated by run-time voting that were not detected by the fault-elimination techniques. If so, then the use of fault elimination does not preclude the use of fault tolerance, i.e., they are complementary techniques rather than competitive techniques. Again, table 3 shows that this did occur for 67 faults, although, as discussed in the rest of this section, the average individual triplet tolerated 33 faults (of the 104 faults present in the average individual triplet) and did not do this consistently. Firm conclusions cannot be drawn from this data given the novice nature of the participants in the fault-elimination efforts, but it does raise interesting questions for further study.

Voting actually did somewhat worse in comparison with testing than is indicated by the data in table 3. In the table, run-time voting is credited with tolerating a fault if it tolerates at least one failure caused by that fault even though it may not tolerate every failure caused by the fault. It is also credited with tolerating a fault if only one or several of the 56 combinations of versions tolerate it even though all

of them do not. The situation is different for fault elimination since the detection of a fault by a fault-elimination technique leads to elimination of the fault prior to production use of the software and elimination of all failures related to that fault. In general, we found that even when the failure caused by a fault is at times tolerated by a triplet, it is usually not tolerated every time, and there is wide variation among the different triplets in terms of how effective they were in tolerating faults.

In order to show the variation, we computed the number of faults tolerated at least once by a triplet divided by the total number of faults that caused a failure in one of the versions comprising that triplet. This fraction ranged from 60.4% to 88.6% with an average of 75.9%[3] and a standard deviation of 6.2%. That is, even the best triplet missed 11% of the faults that it should have been able to tolerate.

Another way of looking at variability among triplets is to consider the conditional probability that a triplet will mask a failure given that a failure occurs (i.e., the conditional probability that a correct result is produced despite the failure of one of the versions). This fraction was even lower, i.e., it ranged from 20.8% to 61.5% with a mean of 37.9% and a standard deviation of 11.1% (see figure 1). On average the triplets only tolerated faults 38% of the time that they caused a failure. This can be explained by the large number of correlated failures that occurred.

4.2 Comparison of Fault-Detection Techniques

Some comparison of the fault-detection techniques is possible with this data, although absolute numbers may not be important because of the problems of evalu-

[3]Note that these are percentages of the faults present in the three versions that constitute the voting triplet and not percentages of all faults found in all versions.

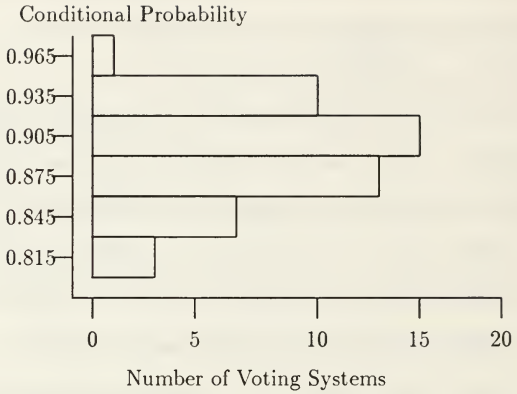

Figure 1: Conditional Probability of Triplet Fault Masking

ating and keeping constant the amount of effort put into each technique. Furthermore, numbers are not really the issue; instead a more important question may be whether different or similar faults are found by each technique. A technique may only find one fault, but if that fault is not likely to be found in any other way, then that technique may still need to be applied.

4.2.1 Definition of Terms

Before discussing the results, certain key concepts need to be clearly defined. In particular, it is important to understand what a fault is and when it may be detected by each technique. The IEEE standard defines a fault to be "an accidental condition that causes a functional unit to fail to perform its required function."[1]

If the correction of a section of code eliminates at least one failure, it is counted as a single fault. Several faults could contribute to the failure for a given data set, and several failures could be due to a single fault. For example, a single data set could reveal separate faults dealing with battalion location and observation of one battalion by another. If either of these faults were solely responsible for failures in other data sets, they are counted as separate faults. If correction of either of these faults eliminates the failure, they are counted as a single fault. This is because faults may sometimes be due to actions distributed through the version code. For example, if a failure results from the initialization code not ensuring an assumption made in some calculation code, this is counted as only one fault, although it could be corrected either by changing the initialization code to ensure the assumption or by changing the calculation code to obviate the assumption.

For most of the techniques used in this experiment, determination of when the techniques detected faults is straightforward. Static analysis and code reading identify the fault precisely to the section of code in which it occurs. A run-time assertion generates reports when faults produce an erroneous run-time state. Testing detects a fault when the test-failure conditions are satisfied due to behavior caused by a fault. The test-failure conditions are explicitly given in the functional test plans and are developed as part of the test data during the structural testing. Faults detected during analysis of the version source code to formulate the structural test data are also considered detected by structural testing.

For back-to-back testing, the conditions when a fault is detected are less clear. We define a fault as detected if the version containing it is identified as erroneous because its answer differs from a majority of the versions. For example, if a triplet

produces Good-Good-Bad results, then the fault responsible for producing the Bad output will be counted as detected. However, Bad-Bad-Good (where the two Bad results are identical) will not count as a fault detection because a fault has not been identified in the incorrect version, but instead the correct version has been erroneously identified as containing a fault. That is, a fault is counted as detected if the voting process isolates the version that is faulty. As another example, Bad1-Bad1-Bad2 would count as only one fault detection (the fault responsible for the Bad2 output) while Bad1-Bad2-Bad3 would count as three faults detected since all three versions would need to be debugged.

It could be argued that Bad1-Bad1-Bad2 would actually result in finding two or three faults because in trying to fix the Bad2 fault, the Bad1 would eventually be found. However, we feel that once a fault is located and fixed in the Bad2 version, the debuggers would probably stop. The same type of argument could be made in the Bad-Bad-Good case where eventually the debuggers might stumble onto the faults causing the Bad results when they gave up attempting to fix the Good program. This seems to us to be overly optimistic. The tendency will be to try to get the single answer to match the multiple answer rather than vice versa. In fact, there were occasions when we temporarily "broke" a correct version when trying to debug it and get it to match the majority result. We felt that to count a fault detection technique as detecting a fault, it should at least identify the program that has failed. This decision is arbitrary, but seemed to make the most sense to the authors.

Two-version voting detects faults if the pair disagree as to the result. Good-Bad is counted as detecting the fault that caused the Bad answer, and Bad1-Bad2 is

counted as detecting the two faults that caused the two distinct bad results. One side effect of this decision is that all faults detected by three-version voting are also (by definition) detected by two-version voting, but the converse is not true.

We assumed that many more test cases can be executed when using back-to-back testing with random generation of test cases because of the lack of necessity to apply an independent validation procedure to the outputs although there is also a necessity to write a test-harness program to implement the voting. Writing a test harness is not necessarily a trivial problem when real numbers are involved since different correct results are possible from different (correct) algorithms due to the use of limited-precision arithmetic. Using a tolerance for the comparisons will not solve the problem [9]. In previous experiments, this consistent comparison problem has resulted in time-consuming debugging of correct programs.

The reader should note that we are now reinterpreting our experimental procedures. In the previous section, we identified the execution of the 10,000 input cases as a simulation of the production use of the software. We are now interpreting this procedure as a fault-elimination technique that would precede the actual production use of the programs. There is no problem with this from a practical standpoint since the procedures are identical and differ only in the time they are performed, but it may be confusing to the reader.

4.2.2 Fault Detection Summary

Table 4 shows the number of faults detected by each technique[4]. The first five lines (those marked 'only') give the number of faults detected by each technique that were detected by none of the other techniques (e.g., run-time assertions detected a total of 23 faults that were not detected by voting, testing, static analysis or code reading). The remainder of the table gives the number of faults detected in common by the techniques named on each line (e.g., code reading found a total of 4 faults that were also found by run-time assertions, but were not found by any other technique).

The voting technique used in constructing table 4 was two-version voting. Three-version voting detected 112 of the 123 faults found by two-version voting. These faults were found by voting with the eight versions combined into the 28 possible pairs and the 56 possible triplets. The values in the line marked 'Voting detection only' table 4 are the maximum and minimum of the faults detected by each two-version voting pair for each version. A range exists for voting because it was applied to each version 7 times (the number of two-version voting systems in which each version participated) while the other techniques were only applied once. In all other voting cases (voting in combination with each of the other techniques), there were no variations in the number of faults detected. The interesting feature of table 4 is not the precise values shown (which depend on the application), but that most of the faults detected by each technique were found by no other technique.

[4]Note that the figures in table 3 and table 4 are not comparable due to: a) the difference between fault detection by voting and fault tolerance by voting and b) the difference between 2-version and 3-version voting

Method	Version 1	2	3	4	5	6	7	8	Total
Testing only	2	12	21	11	13	1	11	10	81
2-Version voting only	$\frac{10}{11}$	$\frac{9}{9}$	$\frac{11}{12}$	$\frac{7}{8}$	$\frac{14}{14}$	$\frac{7}{8}$	$\frac{5}{6}$	$\frac{10}{10}$	$\frac{73}{78}$
Code reading only	0	2	4	2	0	1	0	16	25
Assertions only	3	3	1	8	1	1	3	3	23
Static analysis only	0	0	2	0	0	0	0	0	2
Both 2-v. voting & test	3	1	1	3	2	6	4	0	20
Both assertions & test	5	1	0	2	5	3	0	0	16
Both assertions & 2-v. voting	0	0	2	0	2	0	4	4	12
Both reading & assertions	2	0	0	1	1	0	0	0	4
Both static analysis & 2-v. voting	0	0	0	0	0	1	1	0	2
Both reading & 2-v. voting	0	0	2	0	0	0	0	0	2
Both reading & test	0	0	0	0	0	0	1	0	1
Both static analysis & test	0	0	0	0	0	0	1	0	1
Assert & 2-v. voting & test	0	0	1	1	2	1	0	0	5
Reading & 2-v. voting & test	0	2	0	0	0	0	0	2	4

Table 4: Number of Faults Detected

4.2.3 Variation in Voting Performance

To give some feeling about the variability of the voting performance, two sets of statistics are provided. The first set is the number of faults detected at least once by each pair and each triplet, divided by the total number of faults that caused at least one failure in the versions making up the system (i.e., the fraction of revealed faults that each voting system detected). For the 28 two-version voting systems, the fraction of faults detected varies from 91.3% to 100% with a mean of 97.9% and a standard deviation of 2.6%. For the 56 three-version voting systems, the fraction of faults detected varies among the triplets from 90.5% to 100% with a mean of 96.5% and a standard deviation of 2.5%. In short, the majority of pairs and triplets fail to detect at least some of the faults revealed by the input data. Since virtually all systems would be developed with at most one pair or triplet, the data in table 4 represents a best case for voting.

The second set of statistics used to analyze the variation in fault detection is the conditional probability that a pair or triplet detects each fault given that it is revealed. The mean of these probabilities over all faults detected for each two-version system varies between 0.826 and 0.981, with a mean of 0.934 and a standard deviation of 0.040 (see figure 2). In other words, voting with a two-version system never detected all of the failures of the component versions. Voting with the majority of two-version systems failed to detect 5% or more of the version failures. For the three-version voting systems, the mean of the conditional probabilities over all faults detected by each system varied from 0.787 to 0.953 with a mean of 0.886. (see figure 3). This indicates that there is a considerable chance that three-version back-to-back testing will fail to identify erroneous versions. On the average, this

happened during this experiment in one out of every nine version failures.

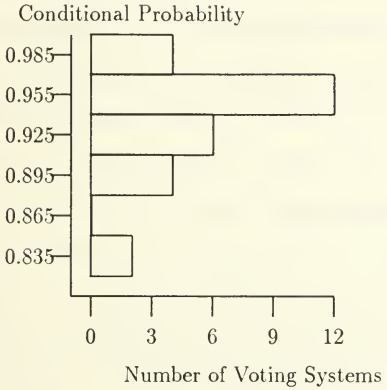

Figure 2: Two-Version Conditional Probability of Detection

4.2.4 Back-To-Back Testing

There are two particularly interesting comparisons to make that deal with currently unresolved issues in testing research. The first is the use of back-to-back testing vs. the use of other testing oracles (i.e., those not involving a voting procedure). Back-to-back testing allows a large amount of data to be executed due to the automated nature of the oracle, and it has been advocated as a way of extensively testing complex software where determining a correct answer by a non-voting procedure may be tedious and time-consuming[7, 8, 28, 31]. Of course, if one takes a larger perspective, part or all of the savings in testing may be offset by the cost

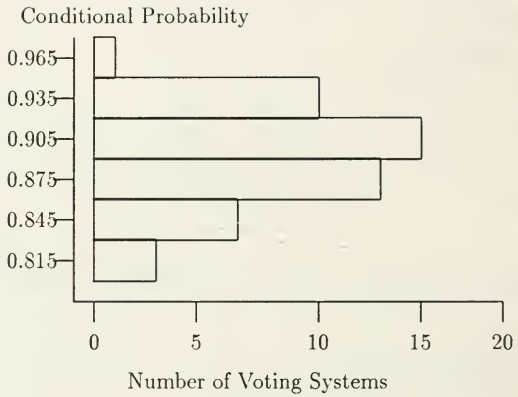

Figure 3: Three-Version Conditional Probability of Detection

of producing multiple versions of the software. However, if back-to-back testing is much more effective then the cost arguments may be irrelevant.

There were 78 faults that were detected by the voting procedure that were not detected by any other technique. Better implementation of the other testing techniques or the use of different testing techniques might have found more faults, but this would have to be proven either way. Even given the novice nature of the participants in the other testing procedures, they did find 153 faults that were not detected by the back-to-back testing. 45 faults were detected in common. There were faults that did not cause failures on the randomly-generated test data and therefore could not possibly have been detected by the back-to-back testing, but were found by the techniques that do not require failure to detect faults.

Our data suggests that using back-to-back testing on randomly-generated data is *not* an acceptable testing procedure by itself. A related question is whether better results are obtained by doing the back-to-back testing on both randomly-generated test cases and functionally-generated test cases. This separates the issue of test data generation from the issue of using voting as a test oracle. We executed the 56 triplets on the functionally-generated and structurally-generated test cases and did not detect any additional faults. This implies that the problem is not necessarily in the test case generation method, but in the identification of errors by voting, i.e., by the limitations of using voting as a test oracle.

4.2.5 Types of Faults Detected

To examine the fault detection behavior of the techniques further, the types of faults detected by each were profiled and compared. Because there is no widely accepted

detailed taxonomy for fault classification, a 13 class fault taxonomy was developed and used. This fault taxonomy is described in table 5. Since this taxonomy was developed to differentiate between the sets of actual faults detected by the various techniques (i.e., developed after examining the faults in an attempt to distinguish them), significance tests on these classifications are inappropriate. In the following discussion, a statement that a technique detected faults in a particular class does not imply that the technique detected all faults of that class. For example, the statement that voting detected missing-thread faults should not be interpreted as indicating that all missing-thread faults located in the versions were detected by voting. Two of the categories in this fault taxonomy (Overrestriction faults and Data-Structure faults) are not mentioned in the text that follows since these categories do not provide a basis for characterizing the differences observed in fault detection by the various techniques.

Code Reading by Stepwise Abstraction

Code reading by stepwise abstraction on uncommented code found calculation faults, missing-check faults, branch-condition faults and missing-branch faults. The participants did not find large global pieces of missing code or missing threads of logic that ran through the entire program.

Analysis of the experiment data lent insight into two questions related to the use of code reading in software development. The first of these questions is what conditions lead code reading to fail to detect faults. While code reading detected a number of faults, it failed to detect the majority of faults. Identification of the causes of this non-detection suggests ways to improve code reading to broaden its effectiveness. One reason for the failure of code reading to detect certain faults was

Class	Comments	Detecting Technique
Overrestriction	E.g., forcing all weather to move north-east, rejecting legal input	Assert, Read, Test, Vote
Loop Condition	E.g., infinite loop	Vote, Assert, Test
Calculation	Incorrect formula	Read
Initialization	Variable not initialized	Stat. Analysis, Test
Substitution	Wrong variable used	Vote, Assert
Missing check	Exceptional case not handled E.g., divide by zero	Read
Branch Condition	Bad condition on a branch	Vote, Read, Test
Missing Branch	Localized missing code to detect and handle specific conditions in normal execution	Read, Test
Missing Thread	Missing path throughout program	Vote, Test
Unimplemented Requirement	Missing functionality on all paths	Test
Ordering	Operations in wrong order (e.g., updating value before use)	Vote, Test
Parameter Reversal	Actual parameter order permuted with respect to formal parameter	Vote, Assert
Data Structure	E.g., linked list becomes circular	Vote, Test, Read, Assert

Table 5: Fault Taxonomy

the omission of needed detail in the abstractions constructed by the participants. Another condition that seemed difficult to detect was missing code. The code-reading participants detected missing-branch faults (i.e., where the set of cases handled by the code did not cover all possible cases at a specific point), but failed to detect those cases where larger or more widespread code was omitted.

A second question of interest in examining the code-reading results is what conditions lead the code-reading participants to erroneously report code as faulty. These erroneous reports seemed to occur when the readers were mislead by the code. Preventing such misdirection may help to improve the detection performance of code reading. Use of commented code for the code reading might have prevented some of this misdirection of the readers, however it might also have lead to different types of misdirection. For example, it is equally possible that the readers would report code as erroneous when it conflicted with the comments (e.g., where the development participants corrected the code, but not the comments describing the code). Another possibility is that the readers would have failed to detect as many faults, e.g., if they summarized the comments instead of the code and thus duplicated the faulty assumptions made by the development participants. In an effort to avoid these problems, as well as the inherent inequality of information based on the widely varying amounts of comments in the version source code (see table 1), all comments were stripped from the versions before the readers were given them.

Analysis of the annotations written by the code readers indicates that false alarms arose from code that was difficult to abstract. For example, an erroneous fault report was generated for a procedure with a large number of arguments that was called in several places in the code. Some of the formal parameters were used in different

ways in the procedure (depending on the value of other formal parameters), and this led to misconceptions on the part of the reader. False alarms also occurred when abandoned implementation strategies (blind alleys during development) are reflected in the code. For example, the readers erroneously reported several faults in cases where the name of a variable conflicts with the manner in which the variable is used. Annotations by the code readers in such situations indicate that they focused on syntactic factors rather than the program semantics. These results suggest that experience and improved training may help reduce erroneous reports from code reading.

Static Data-Flow Analysis

Static data-flow analysis found only initialization faults. Three faults were found by this technique that were not detected by any other. Upon examination, it was determined that the compiler and operating system versions being used happened to initialize to zero the particular storage locations where the programs were loaded, and these variables were used for counters and needed to be initialized to zero. Obviously, this cannot be counted on in future versions of these support programs so these are real and important faults to detect.

Voting

Voting found missing-thread faults, parameter-reversal faults, substitution faults, ordering faults and faults (subsets of loop-condition and data-structure faults) causing abends (which, despite their cause, are obviously found by any of the techniques that involve executing the code over a large number of test cases).

It is interesting to consider the faults that were not found by voting, i.e., those that were so highly correlated that the faults were masked by the voting procedure. For the most part, these were missing-branch faults. This is consistent with past experiments, which have all reported that missing-logic errors are poorly tolerated by multi-version systems. Testing strategies, such as functional and structural testing, that examine special cases as well as typical cases were more successful at finding missing-branch faults. As discussed above, performing back-to-back testing on the test cases derived for functional testing did not solve the problem since the common faults masked the identification of the fault even though the programs failed.

Another unmasked fault involved the use of a wrong subscript. This is puzzling as the same thing happened in a previous experiment [10]. We cannot currently find any other explanation aside from coincidence although we are in the process of attempting to determine if an explanation exists.

Run-Time Assertions

Run-time assertions found parameter-reversal faults, substitution faults and faults causing abends. They did not detect any of the four classes of missing-code faults. We are not very confident about the data for run-time assertions as the programmers involved did not have any experience in writing exception or error-detection code, and our subjective evaluation of their assertions is that

they were, in general, quite poor. All of the run-time assertions used were simple range or specific value tests (e.g., run-time assertions to check if the variable `Params.NumWeatherEvents` lies between 0 and the constant `MaxWeather`, or if pointers are non-nil). In fact, examination of the design documents show cases where the development participants anticipated faults that actually occurred in their code, but (for reasons known only to them) they omitted assertions to check for these faults.

Each pair of development participants added assertions to their version to check if battalions left the simulated battlefield, and these assertions detected the errors that were generated when this occurred. Two of the eight versions placed range restrictions on internally-calculated values, and further faults were detected by these assertions in each of these versions.

The simple check strategy used and the failure by the majority of the development teams to place assertions to check internal results left many errors undetected. However, despite these weaknesses simple range checks detected faults. These checks detected 23 faults that were found in no other manner. The fact that assertions detected faults that voting did not is consistent with the results of our previous study of assertion effectiveness[24]. It appears that even a cursory set of assertions has some value, and this suggests that it would be useful to perform further work to examine the effectiveness of a thorough set of assertions for fault detection (possibly using an automated strategy).

Functional and Structural Testing

Functional and structural testing identified ordering faults, missing-branch faults, unimplemented-requirement faults and missing-thread faults. They also detected

faults causing abends (as did all the techniques that involved executing the programs). Structural testing detected further missing-code faults, in particular faults involving variables that were initialized in a manner that in rare cases conflicted with the manner in which those variables were used.

Structural testing failed to detect several missing-thread faults that were found by other techniques (such as voting). The incompleteness seemed largely due to the module-by-module nature of the testing tool used. That is, the prototype version of ASSET used in this experiment measures the coverage achieved by the input data on each module individually, with no consideration of data flow between modules. The versions contain several instances where global data structures are initialized in one module, updated in a second module and used in calculations in several other modules. While all of the initialization paths and all of the update paths are covered by the test data, not all of the update paths are covered for each initialization path. Therefore, several of the missing-code faults eluded detection in our structural testing.

Examination of the functionally-specified test data sets showed that faults were revealed only by those data sets that contained atypical data (i.e., those tests that exercised special cases in the versions or odd combinations of the functions supported by the code). This result supports a recommended practice in the field of software testing.

Additional Comparisons of Fault Detection Techniques

Two general attributes accounted for much of the observed variation of effectiveness: the ability of the techniques to examine internal states and the scope of their evaluation.

One reason voting failed to detect some faults was that it was not able to examine internal program states. The other techniques do not share this limitation. For code reading and static analysis, examination of the internal state involves evaluation of the program source code. Functional testing identifies and evaluates internal abstract functions. Assertions evaluate specific internal conditions at the locations where they are inserted. Because the voting systems examine only final states, they fail to identify faults that occur, but are concealed by later processing.

Tso and others argue that voting may be performed on internal program states, as in the cross-check analysis technique[32]. However, the programs in this experiment are quite diverse. The internal program states differ significantly in the algorithms and data structures employed. A single value in the internal state of one program may indeed be a single value in another program, but more often it is either a function of several values or not present at all (unneeded in the alternate algorithm used in the second program). Furthermore, since the order of the programs' operations are also quite diverse, there is no single time except initialization and production of the final result at which any correspondence in values could be compared by voting. To allow voting on internal program states requires specification of the algorithm and data structures used in the internal states, effectively eliminating any significant design diversity and thus eliminating the ability to detect design errors.

A second important attribute is the scope of evaluation. The scope over which assertions and code reading examine the system state appears to be the key characteristic limiting the detection of faults by those techniques. Assertions examine the system state at specific points in the execution. If a fault has not yet occurred

at those points, or if the fault's effect is masked, the assertion does not detect the fault. The use of code reading by stepwise abstraction on the uncommented form of these programs did not detect certain faults because the process of abstraction did not maintain sufficient detail. For example, an assumption by the coder might be violated by the faulty code, but the assumption is not preserved through several layers of abstractions made by the reader between the initialization and the calculation code. The purpose of abstraction is to keep the amount of information at a manageable level, but over-abstraction limits the effectiveness of code reading.

5 Conclusions

It is important to consider several caveats when drawing conclusions from the data presented in this paper. First, experts in the various techniques were not used. Students get a lot of experience in programming while in school, but they seldom receive adequate exposure to and practice with testing and other fault-elimination techniques. We gave them training, but that is not a substitute for experience. Furthermore, only one method was applied within each category of fault-elimination techniques; the particular method chosen may not have been the most effective. Finally, our program may not be representative of a large number of applications and the particular software development procedures also may not be representative.

Despite these limitations (which unfortunately are inherent in this type of experimentation), useful information can be derived from this study. In the few instances where there is other experimental evidence, our results tend to support and confirm previous findings. Where almost no experimental evidence is available, our results

represent one data point that can be used to focus and direct future experiments.

This experiment is the first to investigate the relationship between fault-elimination techniques and software fault tolerance. We found that our data does *not* support the hypotheses that multi-version voting is a substitute for functional testing, that testing can be reduced when using this software fault-tolerance technique, nor that testing can proceed in conjunction with operational use of the software in an n-version programming system where high reliability is required. Instead, we found that multi-version voting did not tolerate most of the faults detected by the fault-elimination techniques. Although we also found that multi-version voting tolerated different faults than were detected by the fault-elimination techniques, no firm conclusions should be drawn from this because of doubts about the ability of the novices involved and the limitations of the fault elimination techniques used; further investigation is suggested.

This experiment examines a broad set of fault detection techniques in a comparative manner. While the presence of multiple versions can speed the execution of large numbers of randomly generated cases, our results cast doubt on the effectiveness of using voting as a test oracle. Testing procedures that allow instrumenting the code to examine internal states were much more effective. When comparing fault-elimination methods, we found that the intersection of the sets of faults found by each method was relatively small. Examination of the faults allowed us to categorize the types found by each method and, in some cases, to explain why these results occurred.

This experiment raises questions with respect to several of the techniques examined. The detection capability of code reading appears to be reduced in comparison

to earlier results such as those reported by Basili and Selby[6] (who used smaller programs). Additional research is needed to distinguish the effects of program size and complexity on the effectiveness of code reading. Analysis of the faults not detected shows that there is a need to develop extensions to code reading techniques that better characterize global effects. One way of accomplishing this might be to mix a top-down code reading technique with the bottom-up methodology of code reading by stepwise abstraction. Further investigation of this seems worthwhile.

The static data flow analysis technique used in this study is limited in the type of faults it can potentially detect. However, several of the faults found by this technique were found by no other technique, and so applying it in software development may be worthwhile, particularly given its relatively low cost of application. There may also be language or environmental factors that reduced the number of undefined reference faults in this particular software. For example, the requirement of declaring all variables in Pascal may serve as a reminder to initialize variables before use. Other static analysis techniques, such as associating physical units with variable values and analyzing the software to see if the units are appropriately preserved[21] deserve further exploration. These types of static analysis techniques would permit examination of the legality of usage rather than just the presence of initialization and reference.

6 Acknowledgement

The authors are pleased to acknowledge the participants involved in this experiment: David Bainbridge, Julie Blaes, Debra Brodbeck, Trinidad Chacon, Un-Young Choi,

Emil Damavandi, Daniel Dayton, David Djujich, Robert Fergurgur, Samuel Horrocks, Julius Javellana, Debra Johnson, Erick Jordan, Nils Kollandsrud, Douglas Labahn, So-kang Liu, Tod Mauerman, Michael Mellenger, Caroline Nguyen, Tuan Nguyen, Brad Nold, Blaise O'Brien, Kenneth Oertle, Stephen Omnus, Pauline Otah, Ya-Ching Pan, Gary Petersen, David Stapleton, Victor Tam, Hien Tang, Christina Tranhuyen, Brian Watson, Penny Whitsitt, and Mark Womack. Stephanie Leif aided in many different aspects throughout the work described here. George Dinsmore and Robin Kane of TRW corporation were most helpful in the supervision of the experiment. Phyllis Frankl and Elaine Weyuker provided the ASSET testing tool. Debra Richardson and Richard Selby provided many constructive comments.

References

[1] "Glossary of Software Engineering Terminology", ANSI-IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[2] T. Anderson, P.A. Barrett, D.N. Halliwell and M.R. Moulding, "Software Fault Tolerance: An Evaluation", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 1502–1510.

[3] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software", *Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 1491–1501.

[4] A. Avizienis and J.P.J. Kelly "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, August 1984, pp. 67–80.

[5] A. Avizienis, M.R. Lyu and W. Shütz, "In Search of Effective Diversity: A Six-Language Study of Fault Tolerant Flight Control Software", *International Symposium on Fault Tolerant Computing Systems*, Tokyo, Japan, June 1988, pp. 15–22.

[6] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, December 1987, pp. 1278–1296.

[7] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahl and J. Lahti, "PODS – A Project on Diverse Software", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, (1986), pp. 929-940.

[8] S.S. Brilliant, *Testing Software Using Multiple Versions*, Ph.D. Dissertation, University of Virginia, Charlottesville, VA, September 1987.

[9] S.S. Brilliant, J.C. Knight and N.G. Leveson, "The Consistent Comparison Problem in Multi-Version Software", *IEEE Transactions on Software Engineering*, in press.

[10] S.S. Brilliant, J.C. Knight and N.G. Leveson, "Analysis of Faults in an N-version Software Experiment", in press.

[11] J.E. Brunelle and D.E. Eckhardt "Fault Tolerant Software: Experiment with the SIFT Operating System," *AIAA Computers in Aerospace V Conference*, October 1985, pp. 355–360.

[12] L. Chen and A. Avizienis, "N-Version Programming: A Fault Tolerance Approach to the Reliability of Software", *Eighth Int. Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1978, pp. 3–9.

[13] A. W. Dobieski, "Modeling Tactical Military Operations", *Quest*, Spring 1979, pp. 1–25.

[14] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability", *ACM Computing Surveys*, Vol. 8, No. 3, September 1976, pp. 305–330.

[15] P. Frankl and E. Weyuker, "Data Flow Testing in the Presence of Unexecutable Paths", *Workshop on Software Testing*, Banff, Canada, July 1986, pp. 4–13.

[16] M.R. Girgis and M. R. Woodward, "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria", *Proceedings of the Software Testing Workshop*, Banff, Canada, 1986, pp. 64–73.

[17] L. Gmeiner and U. Voges, "Software Diversity in reactor protection systems: an experiment", *SAFECOMP 1979*, Frankfurt, Germany, May 1979, pp. 75–79.

[18] G. Hagelin, "ERICSSON Safety System for Railway Control" in *Software Diversity in Computerized Control Systems*, U. Voges (ed.), Springer-Verlag, New York, 1987, pp. 11–21.

[19] W.C. Hetzel, *An Experimental Analysis of Program Verification Methods*, Ph.D. Dissertation, University of North Carolina at Chapel Hill, 1976.

[20] W.E. Howden, "Functional Testing and Design Abstractions", *Journal of Systems and Software*, Vol 1, 1980, pp. 307–313.

[21] W.E. Howden, "A Survey of Static Analysis Methods", *Tutorial: Software Testing and Validation Techniques*, IEEE Press, 1981, pp. 101–115.

[22] J.C. Knight and N.G. Leveson, "Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Transactions on Software Engineering*, January 1986, pp. 96–109.

[23] J.C. Knight and N.G. Leveson, "An Empirical Study of Failure Probabilities in Multi-Version Software," *Sixteenth Int. Symposium on Fault-Tolerant Computing*, Vienna, Austria, July 1986, pp. 165–170.

[24] N.G. Leveson, S.S. Cha, J.C. Knight and T.J. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study",1989 (Submitted for publication).

[25] R.C. Linger, H.D. Mills and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979, pp. 147–212.

[26] P.M. Melliar-Smith and R.L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight-Control System," *IEEE Trans. on Computers*, Vol. C-31, no. 7, July 1982, pp. 616–630.

[27] G.J. Myers, "A Controlled Experiment in Program Testing and Code Walk-Throughs/Inspections," *Communications of the ACM*, Sept. 1978, pp. 760–768.

[28] C.V. Ramamoorthy, Y.K., Mok, E.B. Bastani, G.H. Chin and K. Suzuki, "Application of a Methodology for the Development and Validation of Reliable Process Control Software," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 6, November 1981, pp. 537–555.

[29] C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 46–58.

[30] J.C. Rouquet and P.J. Traverse, "Safe and Reliable Computing on board the Airbus and ATR Aircraft", *SAFECOMP 1986*, Sarlat, France, October 1986, pp. 93–97.

[31] F. Saglietti and W. Ehrenberger, "Software Diversity — Some Considerations about its Benefits and its Limitations," *Safecomp '86*, Sarlat, France, October 1986.

[32] K.S. Tso, A. Avizienis and J.P.J. Kelly, "Error Recovery in Multi-Version Software Development", *Safecomp '86*, Sarlat, France, October 1986, pp. 43–50.

Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Library, Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 4401 Ford Ave. Alexandria, VA 22302-0268	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Computer Science Department Code 52 Naval Postgraduate School Monterey, CA 93943	1
Stephanie Aha ICS Department University of California, Irvine Irvine, CA 92717	1
Thomas Anderson Computing Centre University of Newcastle-upon-Tyne 6 Kensington Terrace Newcastle-upon-Tyne, NE1 7RU, England	1
Victor Basili Dept. Comput. Sci Univ. Maryland College Park, MD 20742	1
Robin E. Bloomfield Adelard 28 Rhondda Grove London, England E3 5AP	1

Stephen Cha
ICS Department
University of California, Irvine
Irvine, CA 92717

Lori Clark
Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, Massachusetts 01002

Barry Daniels
Manager, Software Engineering: Systems
National Computing Centre Limited
Oxford Road
Manchester, England M1 7ED

George Dinsmore
TRW
1 Space Park 134/3816
Redondo Beach, CA 90278

Wolfgang Ehrenberger
Fachhochschule
Fachbereich Informatik
6400 Fulda, West Germany

George B. Finelli
National Aeronatics and Space Administration
Langley Research Center
Mail Code 130
Hampton, Virginia 32665-5225

Phyllis Frankl
Polytechnic University
333 Jay Street
Brooklyn, New York 11201

Susan Gerhardt
MCC Software Technology Program
3500 W. Balcones Dr.
Austin TX 78759

James Hester
Department of Computer Science
California State University, Fullerton
Fullerton, CA 92634

Werner Hueber 1
Code 3192
Naval Weapons Center
China Lake, CA 92555-6001

John Knight 1
Software Productivity Consortium
Reston, VA 22091

Herman Kopetz 1
Institut fuer Technische Informatik
Technische Universitaet Wien
Gusshausstrasse 30
A-1040 Wien Austria

Nancy Leveson 10
ICS Department
University of California, Irvine
Irvine, CA 92717

Bev Littlewood 1
Center for Software Reliability
City University
Northampton Square
London ECIV OHB, England

Bonnie Melhart 1
ICS Department
University of California, Irvine
Irvine, CA 92717

Peter Neumann 1
SRI International
Menlo Park, CA 94025

David Parnas 1
Computing and Information Science
Queen's University at Kingston
Kingston, Ontario, Canada K7L 3N6

Debra Richardson 1
ICS Department
University of California, Irvine
Irvine, CA 92717

Johann Schumann 1
Forschungsgruppe Kuenstliche Intelligenz
Institut fuer Informatik
Technische Universitaet Muenchen
Augustenstrasse 46/II
8000 Muenchen 2, West Germany

Richard Selby 1
ICS Department
University of California, Irvine
Irvine, CA 92717

Richard Shimeall 1
M/A-Com Linkabit
3033 Science Park Road
San Diego, CA 92121

Stephen C Shimeall 1
Boeing Aerospace
PO Box 3999 M/S 2f-13
Seattle, Wa. 98124-2499

Timothy J. Shimeall 20
Computer Science Dept.
Code 52Sm
Naval Postgraduate School
Monterey CA 93943

J. R. Taylor 1
Head of Institute
Institute for Technical Systems Analysis
Jernbanegade 52 A
DK 400 Roskilde DENMARK

Jefferey C. Thomas 1
Mail Station M1/108
Computer Security Office
Systems and Computer Engineering Division
The Aerospace Corporation
2350 East El Segundo Blvd.
El Segundo, CA 90245-4691

Robert E. Westbrook 1
Code 31C
Embedded Computer Technology Office
Naval Weapons Center
China Lake, CA 92555-6001

Elaine Weyuker
Department of Computer Science
Courant Institute of Mathematical Sciences
New York, NY 10012